**姓名: 张景耀 学号: 3200602123 班级: 计算机2005**

### 实验内容1. 在顺序表中设计函数实现以下操作：

（a）从顺序表中删除具有最小值的元素（假设顺序表中元素都不相同），并由函数返回被删元素的值，空出的位置由最后一个元素填补。

（b）从顺序表中删除具有给定值e的所有元素。

（c）在一个顺序表中如果一个数据值有重复出现，则留下第一个这样的数据值，并删除其他所有重复的元素，使表中所有元素的值均不相同。

#### 源代码

#include<bits/stdc++.h>

using namespace std;

const int DEFAULT\_SIZE = 1e+3;

template<class ElemType>

class SeqList

{

protected:

    int len, maxLen;

    ElemType\* data;

public:

    SeqList(int size = DEFAULT\_SIZE);

    SeqList(ElemType\* data, int n, int size = DEFAULT\_SIZE);

    //从顺序表中删除具有最小值的元素, 并返回最小值, 空位由最后一位填补, O(n)

    ElemType deleteMin();

    //从st开始, 删除全部值为v的元素, O(n^2)

    void deleteAll(ElemType v, int st = 0);

    //去重, 时间复杂度O(n^2)(扫描O(n), 删除O(n)), 哈希表额外需要空间复杂度O(m)(m为不重复的元素个数)

    void unique();

    void traverse();

    virtual ~SeqList();

};

template<class ElemType>

SeqList<ElemType>::SeqList(int size)

{

    this->len = 0;

    this->maxLen = size;

    this->data = new ElemType[size];

}

template<class ElemType>

SeqList<ElemType>::SeqList(ElemType data[], int n, int size)

{

    this->data = new ElemType[size];

    this->len = n;

    for(int i = 0; i < n; i++)

        this->data[i] = data[i];

    this->maxLen = size;

}

template<class ElemType>

ElemType SeqList<ElemType>::deleteMin()

{

    //寻找最小值下标

    int p = 0;

    for(int i = 1; i < this->len; i++)

        if(this->data[i] < this->data[p])

            p = i;

    //用最后一个元素覆盖最小值

    int ans = this->data[p];//记录一下最小值便于返回

    this->data[p] = this->data[this->len - 1];

    this->len--;

    return ans;

}

template<class ElemType>

void SeqList<ElemType>::deleteAll(ElemType v, int st)

{

    for(int i = st; i < this->len; i++)

        //遇到值等于v的元素就删除该元素

        if(this->data[i] == v)

        {

            for(int j = i + 1; j < this->len; j++)

                this->data[j - 1] = this->data[j];

            this->len--;

            //删除之后就自动迭代了一次, 无需更新i

            i--;

        }

}

template<class ElemType>

void SeqList<ElemType>::unique()

{

    unordered\_map<ElemType, bool> hash;

    for(int i = 0; i < this->len; )

        if(hash[this->data[i]])

            this->deleteAll(this->data[i], i);

        else

            hash[this->data[i++]] = true;

}

template<class ElemType>

void SeqList<ElemType>::traverse()

{

    for(int i = 0; i < this->len; i++)

        cout << this->data[i] << " ";

    printf("Current Len is: %d\n", this->len);

    puts("");

}

template<class ElemType>

SeqList<ElemType>::~SeqList()

{

    delete[] this->data;

}

#### 测试函数

int main()

{

    int arr[] = {3, 3, 3, 1, 2, 2, 2, 7, 2, 6, 6, 2, 5, 6, 2, 5, 7, 3, 3, 5, 4};

    SeqList<int> s(arr, 21);

    cout << "The Original SeqList is: " << endl;

    s.traverse();

    int ret = s.deleteMin();

    printf("The Minimum is %d\n", ret);

    cout << "After Delete Min: " << endl;

    s.traverse();

    s.deleteAll(2);

    cout << "After Delete All Elem '2'" << endl;

    s.traverse();

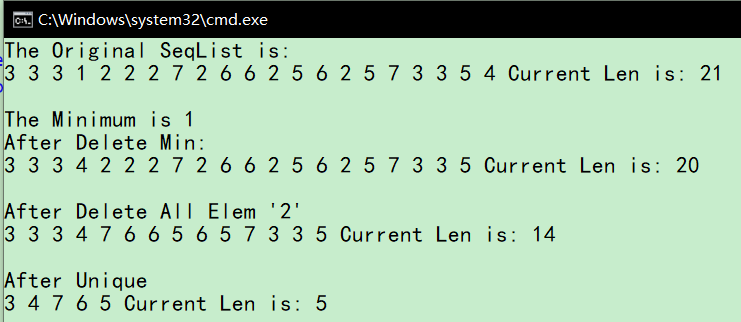
    s.unique();

    cout << "After Unique" << endl;

    s.traverse();

}

#### 测试结果



### 实验内容2. 设计一个有序顺序表类，即表中的数据元素按数据元素值递增有序。实现以下函数：

（a）把给定值e插入有序表中。

（b）删除值为e的所有数据元素。

（c）合并两个有序表，得到一个新的有序表。

（d）从有序顺序表中删除其值在给定值s与t之间（s＜t）的所有元素，如果s≥t或顺序表为空，则显示出错信息，并退出运行。

#### 源代码

/\*

（2）设计一个有序顺序表类，即表中的数据元素按数据元素值递增有序。实现以下函数：

    （a）把给定值e插入有序表中。

    （b）删除值为e的所有数据元素。

    （c）合并两个有序表，得到一个新的有序表。

    （d）从有序顺序表中删除其值在给定值s与t之间（s＜t）的所有元素，如果s≥t或顺序表为空，则显示出错信息，并退出运行。

\*/

#include<bits/stdc++.h>

using namespace std;

const int DEFAULT\_SIZE = 1e+2;

template<class ElemType>

class OrderedList

{

public:

    int len, maxLen;

    ElemType\* data;

public:

    OrderedList(int size = DEFAULT\_SIZE);

    //二分查找表中大于等于v的第一个位置, O(logn)

    //工具函数, 服务其他成员方法

    int find(ElemType v);

    //把给定值v插入有序表中, 利用二分查找O(logn)

    void insert(ElemType v);

    //删除全部值为v的元素, 利用二分查找O(nlogn)(删除复杂度O(n))

    void deleteAll(ElemType v);

    //归并另外一个新表, 并返回该新表

    OrderedList\* merge(const OrderedList& o);

    //删除[s, t]范围内的所有数, 利用二分查找O(nlogn)(删除复杂度O(n))

    void deleteS2T(ElemType s, ElemType t);

    //遍历, O(n)

    void traverse();

    virtual ~OrderedList();

};

template<class ElemType>

OrderedList<ElemType>::OrderedList(int size)

{

    this->data = new ElemType[size];

    this->maxLen = size;

    this->len = 0;

}

template<class ElemType>

void OrderedList<ElemType>::traverse()

{

    for(int i = 0; i < this->len; i++)

        cout << this->data[i] << " ";

    printf("Current Len is: %d\n", this->len);

    puts("");

}

//二分查找第一个大于等于v的位置

template<class ElemType>

int OrderedList<ElemType>::find(ElemType v)

{

    int l = 0, r = this->len - 1, mid;

    //二分

    while(l < r)

        if(this->data[mid = l + r >> 1] >= v)

            r = mid;

        else

            l = mid + 1;

    //l就是答案

    return l;

}

template<class ElemType>

void OrderedList<ElemType>::insert(ElemType v)

{

    if(this->len + 1 > this->maxLen)

    {

        cout << "The List is Already Full!!!" << endl;

        return;

    }

    //l就是要插入的位置

    int l = this->find(v);

    //尾插要Special Judge

    if(this->data[l] < v)

    {

        this->data[this->len++] = v;

        return;

    }

    //把l后的元素全部后移, 给l腾出空间

    for (int i = len - 1; i >= l; i--)

        this->data[i + 1] = this->data[i];

    //插入

    this->data[l] = v;

    //更新长度

    this->len++;

}

template<class ElemType>

void OrderedList<ElemType>::deleteAll(ElemType v)

{

    int l = this->find(v);

    //找不到就返回

    if(this->data[l] != v)

        return;

    int r = l;

    while(r < this->len and this->data[r] == v)

        r++;

    int offset = r - l;

    while(r < len)

        this->data[l++] = this->data[r++];

    this->len -= offset;

}

template<class ElemType>

OrderedList<ElemType>\* OrderedList<ElemType>::merge(const OrderedList& o)

{

    //新建一个有序表

    auto ans = new OrderedList(this->len + o.len + DEFAULT\_SIZE);

    //双指针归并, O(m + n)

    int i = 0, j = 0;

    for(; i < this->len and j < o.len; ans->len++)

            ans->data[ans->len] = this->data[i] < o.data[j] ? this->data[i++] : o.data[j++];

    while(i < this->len)

        ans->data[ans->len++] = this->data[i++];

    while(j < o.len)

        ans->data[ans->len++] = o.data[j++];

    return ans;

}

template<class ElemType>

void OrderedList<ElemType>::deleteS2T(ElemType s, ElemType t)

{

    if(s >= t or !this->len)

    {

        cout << "Error!!!" << endl;

        exit(1);

    }

    //l是第一个大于等于s的位置

    int l = this->find(s);

    //s太大, 无需要删除元素, 直接返回

    if(this->data[l] < s)

        return;

    //寻找区间末尾

    int r = l;

    while(this->data[r] <= t)

        r++;

    //删除

    int offset = r - l;

    while(r < this->len)

        this->data[l++] = this->data[r++];

    this->len -= offset;

}

template<class ElemType>

OrderedList<ElemType>::~OrderedList()

{

    delete[] this->data;

}

#### 测试函数

int main()

{

    OrderedList<int> o;

    srand((unsigned)time(NULL));

    cout << "The Original data Generated Randomly: " << endl;

    for(int i = 0; i < 20; i++)

    {

        int t;

        cout << (t = rand() % 50) << " ";

        o.insert(t);

    }

    cout << "\n\n";

    cout << "Insert Them All into the List: " << endl;

    o.traverse();

    printf("After Delete All Elem '%d':\n", o.data[5]);

    o.deleteAll(o.data[5]);

    o.traverse();

    OrderedList<int> t;

    for(int i = 0; i < 10; i++)

        t.insert(i);

    cout << "When Merged with a Sequence ranged[0, 9]" << endl;

    auto ret = o.merge(t);

    ret->traverse();

    delete ret;

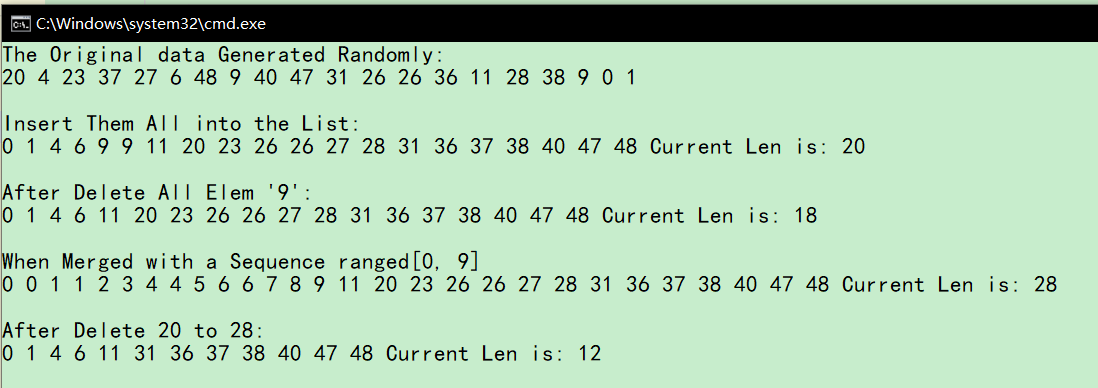
    printf("After Delete %d to %d:\n", o.data[5], o.data[10]);

    o.deleteS2T(o.data[5], o.data[10]);

    o.traverse();

}

#### 测试结果



### 实验内容3. 针对带头结点的单链表，试编写下列函数：

（a）定位函数：在单链表中寻找第i个结点。若找到，则返回第i个结点的地址，否则返回空指针。

（b）统计函数：统计单链表中等于给定值e的元素个数。

#### 源代码

#include<bits/stdc++.h>

using namespace std;

//结点类

template<class ElemType>

class Node

{

public:

    //next指针

    Node<ElemType>\* ne;

    //数据域

    ElemType val;

public:

    Node<ElemType>(ElemType val = 0);

};

//链表类

template<class ElemType>

class LinkedList

{

public:

    //头结点指针

    Node<ElemType>\* head;

public:

    LinkedList();

    //头插

    void insertAtHead(ElemType val);

    //定位函数：在单链表中寻找第i个结点。若找到，则返回第i个结点的地址，否则返回空指针, O(n)

    Node<ElemType>\* find(int i);

    //统计函数：统计单链表中等于给定值e的元素个数, O(n)

    long long statistics(ElemType val);

    //遍历打印函数, 便于调试, O(n)

    void traverse();

    ~LinkedList();

};

template<class ElemType>

LinkedList<ElemType>::LinkedList()

{

    this->head = new Node<ElemType>();

}

template<class ElemType>

void LinkedList<ElemType>::insertAtHead(ElemType val)

{

    Node<ElemType>\* p = new Node<ElemType>(val);

    p->ne = head->ne;

    this->head->ne = p;

}

template<class ElemType>

void LinkedList<ElemType>::traverse()

{

    auto p = this->head->ne;

    while(p)

    {

        cout << p->val << " ";

        p = p->ne;

    }

    puts("");

}

template<class ElemType>

Node<ElemType>\* LinkedList<ElemType>::find(int i)

{

    auto p = this->head->ne;//p初始化为头结点的指针域

    //p迭代i次, 若期间其为空则说明链表长度不足

    while(i--)

    {

        if(!p)

            return NULL;

        p = p->ne;

    }

    return p;

}

template<class ElemType>

long long LinkedList<ElemType>::statistics(ElemType val)

{

    auto p = this->head->ne;

    long long ans = 0;

    //找到值为val的结点就更新ans

    while(p)

    {

        if(p->val == val)

            ans++;

        p = p->ne;

    }

    return ans;

}

template<class ElemType>

LinkedList<ElemType>::~LinkedList()

{

    //析构时要有临时指针记录当前要析构的结点

    auto p = this->head;

    while(p)

    {

        auto t = p;

        p = p->ne;

        delete t;

    }

}

template<class ElemType>

Node<ElemType>::Node(ElemType val)

{

    this->val = val;

    this->ne = NULL;

}

#### 测试函数

int main()

{

    srand((unsigned)time(NULL));

    LinkedList<int> l;

    for(int i = 0; i < 30; i++)

        l.insertAtHead(rand() % 20);

    cout << "The Original LinkedList is: " << endl;

    l.traverse();

    puts("");

    cout << "The 10th Node's Address: " << endl;

    cout << l.find(10) << endl;

    puts("");

    cout << "The 0xfffffth Node's Address: " << endl;

    cout << l.find(0xfffff) << endl;

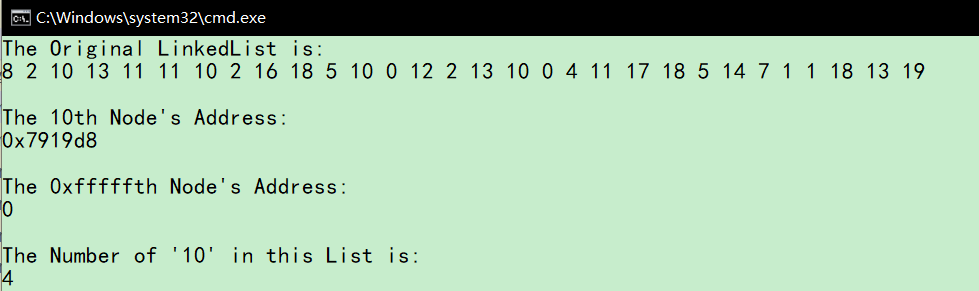
    puts("");

    cout << "The Number of '10' in this List is: " << endl;

    cout << l.statistics(10) << endl;

}

#### 测试结果



### 实验内容4. 设计一个带头结点的有序单链表类。实现以下函数：

（a）插入函数：把元素值e作为数据元素插入表中。

（b）删除函数：删除数据元素等于e的结点。

#### 源代码

#include<bits/stdc++.h>

using namespace std;

template<class ElemType>

class Node

{

public:

    Node<ElemType>\* ne;

    ElemType val;

public:

    Node<ElemType>(ElemType val = 0);

};

template<class ElemType>

class OrderedLinkedList

{

public:

    Node<ElemType>\* head;

public:

    OrderedLinkedList();

    //插入函数：把元素值val作为数据元素插入表中, O(n)

    void insert(ElemType val);

    //删除函数：删除数据元素等于val的结点, O(n)

    void deleteVal(ElemType val);

    //遍历方便调试, O(n)

    void traverse();

    ~OrderedLinkedList();

};

template<class ElemType>

void OrderedLinkedList<ElemType>::traverse()

{

    auto p = this->head->ne;

    while(p)

    {

        cout << p->val << " ";

        p = p->ne;

    }

    puts("");

}

template<class ElemType>

OrderedLinkedList<ElemType>::OrderedLinkedList()

{

    this->head = new Node<ElemType>();

}

template<class ElemType>

void OrderedLinkedList<ElemType>::insert(ElemType val)

{

    auto p = head;

    //寻找第一个数据域大于等于val的结点的前一个结点

    while(p->ne and p->ne->val < val)

        p = p->ne;

    //找到之后进行插入操作

    auto t = new Node<ElemType>(val);

    t->ne = p->ne;

    p->ne = t;

}

template<class ElemType>

void OrderedLinkedList<ElemType>::deleteVal(ElemType val)

{

    auto p = this->head;

    while(p->ne)

    {

        //寻找第一个数据域等于val的结点的前一个结点

        if(p->ne->val == val)

        {

            auto t = p->ne;

            p->ne = p->ne->ne;

            delete t;

            //删除成功后相当于自动迭代了一次, 无需更新p

            continue;

        }

        p = p->ne;

    }

}

template<class ElemType>

OrderedLinkedList<ElemType>::~OrderedLinkedList()

{

    auto p = this->head;

    while(p)

    {

        auto t = p;

        p = p->ne;

        delete t;

    }

}

template<class ElemType>

Node<ElemType>::Node(ElemType val)

{

    this->val = val;

    this->ne = NULL;

}

#### 测试函数

int main()

{

    srand((unsigned)time(NULL));

    OrderedLinkedList<int> l;

       cout << "The Original data Generated Randomly: " << endl;

    for(int i = 0; i < 30; i++)

    {

        int t;

        cout << (t = rand() % 25) << " ";

        l.insert(t);

    }

    cout << "\n\n";

    cout << "After Insert them All The Original OrderedLinkedList is: " << endl;

    l.traverse();

    puts("");

    printf("After Delete %d:\n", 15);

    l.deleteVal(15);

    l.traverse();

}

#### 测试结果

